下载APP

23 | Spanner（二）：时间的悖论

2021-11-22 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 21:42 大小 19.89M

你好，我是徐文浩。

在上节课里，我们一起了解了 Spanner 的整体架构。Spanner 的整个架构并不会让人有什么意外之喜，遵循的仍然是标准的分布式数据库的架构设计，通过对于数据分区、Paxos 同步复制等一系列的机制来实现一个超大规模的全球数据库。而对于网络延时，则是选择了对数据分区进行“调度”的策略，让数据尽可能接近读写它的用户，而不是让一份数据在所有的 Zone 里出现一份。

不过，其实对于 Spanner 这样的系统来说，**最有挑战的问题**并不是如何调度数据，而是在这样一个“全球”数据库里，如何实现事务。这个问题，也就是我们这节课的主题了。这节课，我会主要带你学习这两个知识点：

分布式数据库，会面临哪些我们意想不到的挑战。特别是其中不可靠的网络和不可靠的时钟，会为我们实现事务带来哪些困扰。

为什么 Megastore 这样简单的两阶段事务难以解决这些挑战，而 Spanner 是如何设计机制，来解决这个问题的。

相信学完了这节课，你能对 Spanner 面临的时钟误差下的可线性化挑战有所了解，也能对分布式数据库事务的实现有更深一步的掌握。

Megastore 的事务性能怎么样？

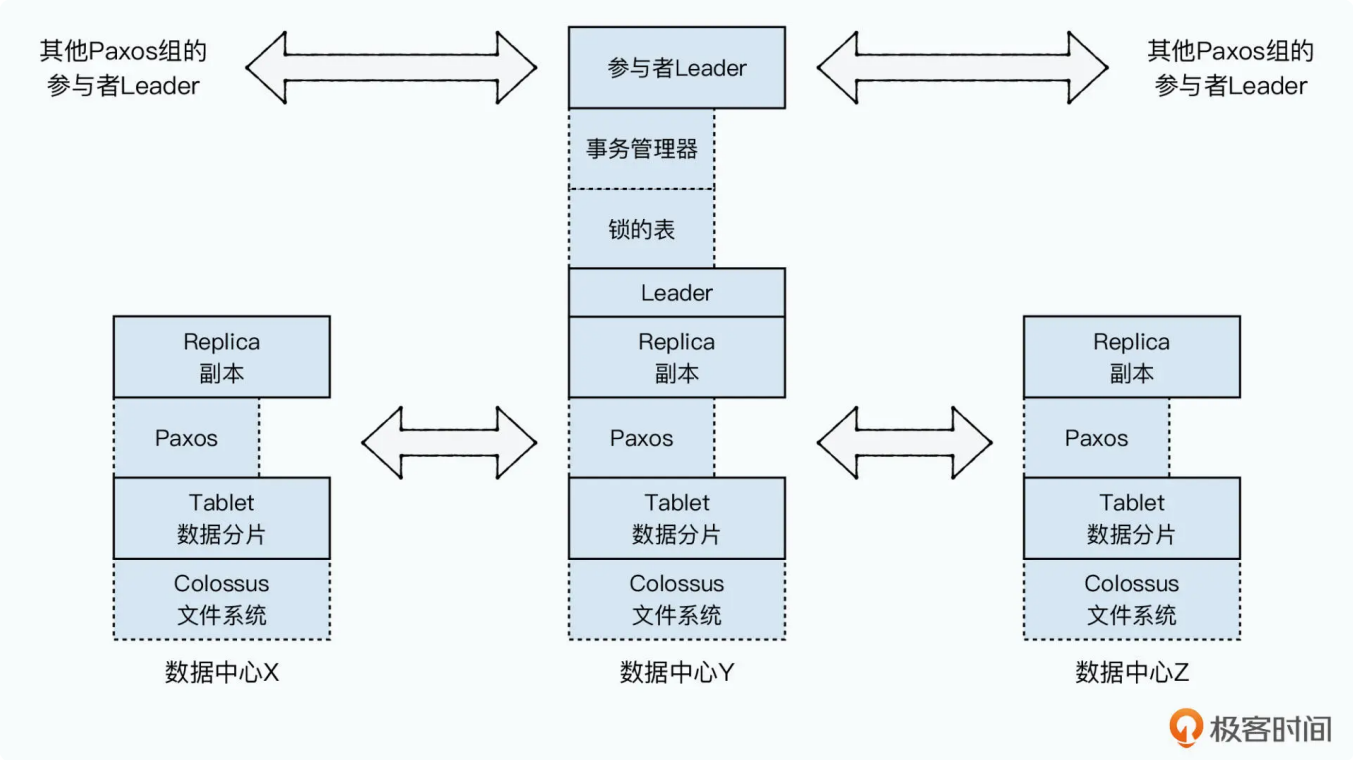
我们先来回顾下 Megastore 的数据库事务实现。在 Megastore 里，我们只能在单个实体组上实现一阶段事务。一旦需要跨越两个实体组，我们要么放弃事务性，采用 Megastore 的消息机制，要么我们就要选择两阶段事务。

事实上，在真实的数据库应用里，我们不太可能避开这里的“两阶段事务”。最典型的，就是所有和“钱”有关的操作。我们在讲解Chubby 论文的时候，就讲过银行转账的例子，我们把 A 的钱转给 B，如果用 Megastore，A 和 B 各自的银行账户，肯定各是一个独立的实体组，而转账，我们显然不能使用所谓的异步消息机制，那么自然，可以选的就是两阶段事务了。

不过，这个时候，问题就来了。我们知道，两阶段事务需要一个“协调者”，**这个“协调者”应该是谁呢？**我们应该用 A 所在的 Paxos 的 Leader 作为协调者吗？还是用 B 的呢？还是我们有一个整个 Spanner 宇宙的协调者？或者我们应该直接用客户端，来作为协调者？如果同时，B 也向 C 发起了转账，C 也向 A 发起转账，这两个事务的协调者又是谁呢？这三个事务并发的情况下，我们该怎么解决事务里的冲突呢？

首先，我们不可能让客户端作为协调者，因为每一个事务请求都可能来自不同的客户端。客户端不知道有哪些并发的事务，自然也无法解决并发冲突问题。其次，我们也不可能选择一个 Spanner 宇宙级别的协调者，无论是并发请求数、还是网络延时，都会打爆这个方案。

所以，**最可行的方案，就是让 Paxos 的 Leader 作为协调者出现。**同时，当这个事务，需要跨越不同的 Paxos Group 的时候，我们还需要两个 Paxos 的 Group 之间做一些协调工作，来解决潜在的并发冲突问题。



Spanner 的架构就是这样的，**每个 Spanserver 上，会有一个事务管理器，用来支持分布式事务**。这个事务管理器，就是一个参与者 Leader（Participant Leader），这个参与者 Leader 会和其他的参与者 Leader 协商，来完成事务的两阶段提交。每个 Spanserver 里，都有一张锁表（Lock Table），在需要实现事务的时候，我们就从锁表里面获取锁。

如果是一个两阶段的跨 Paxos Group 的事务，那么我们就需要通过前面的事务管理器，和其他 Paxos Group 里的事务管理器互相协调，来实现一个两阶段提交。而如果这个事务只在当前的 Paxos Group 里就可以完成，我们就不需要通过事务管理器，直接从锁表里面去获取锁，在本地通过 Paxos 算法进行事务日志的提交就好了。

整个事务管理器的状态，也会通过存储到下面的 Paxos 组里，自然也会复制到多个副本中，所以事务管理器里的事务状态，也是持久化并且高可用的。

分布式事务下的可线性化

乍这么一看，似乎我们就是对每个 Paxos 组上面，给了一个事务管理器，来进行跨 Paxos 组的两阶段事务协调就解决问题了。事情真的就这么简单吗？当然不是了，我们来看看当有两个事务并发的时候，问题会变成什么样子。

我们现在有这样两个事务，一个是张三转账给李四，另一个是李四转账给王二。恰好呢，张三、李四、王二这三个人的数据，并不在一个 Paxos 组里，他们所在的 Paxos 组的

Leader 呢，也不是同一个 Zone。我们假设张三、李四，王二，分别就在 A、B、C 三个 Zone 里，就像表格里给出的这样：

用户	数据副本的Leader所在的Zone	账户余额	事务
张三	A	1000	转账500给李四
李四	B	500	转账300给王二
王二	C	700	转账600给张三

极客时间

事务一，张三账户里有 1000 元，要转账 500 给李四。

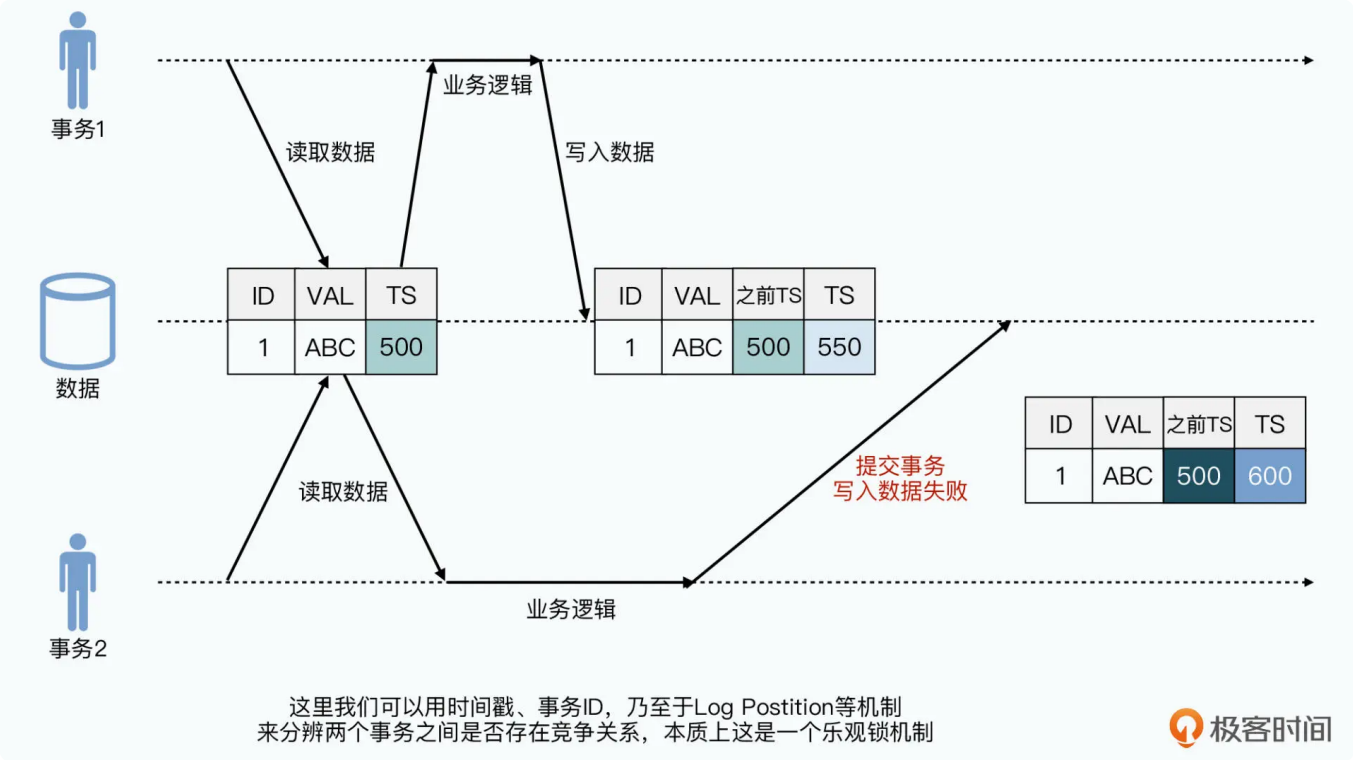
事务二，李四账户里有 500，要转账 300 给王二。

事务三，王二账户里有 700，要转账 600 给张三。

我们要支持数据库事务，所以张三转账 500 给李四这个动作，需要一个分布式事务，也就是 A 和 B 所在的 Paxos 组，需要同时更改张三和李四的数据。而李四转账给王二，同样也需要一个分布式事务，同时修改李四和王二的数据。王二转账给张三也是一样的，需要同时修改王二和张三的数据。

但是因为三个用户不在一个 Zone 里，所以三个事务会从三个不同的 Leader 发起，那么我们就需要解决**分布式数据库的事务并发问题**。我们在 Chubby 和 Megastore 的论文里，已经讲解过分布式事务上的“可串行化”以及“可线性化”的问题，对于 Spanner 这样的系统，其实我们的要求也是一样的。

为了保障可串行化，我们需要给数据打上“版本”，然后多个事务并发的时候。确保读取的数据，是最新版本的快照。然后在事务提交的时候，判断你读写的数据，是否已经有新的版本了，来避免错误地覆盖了已经更新的数据，这也就是所谓的“乐观锁”的并发机制。



在Megastore里，我们已经看过数据库的MVCC并发机制了，我们可以直接使用“乐观加锁”的方式

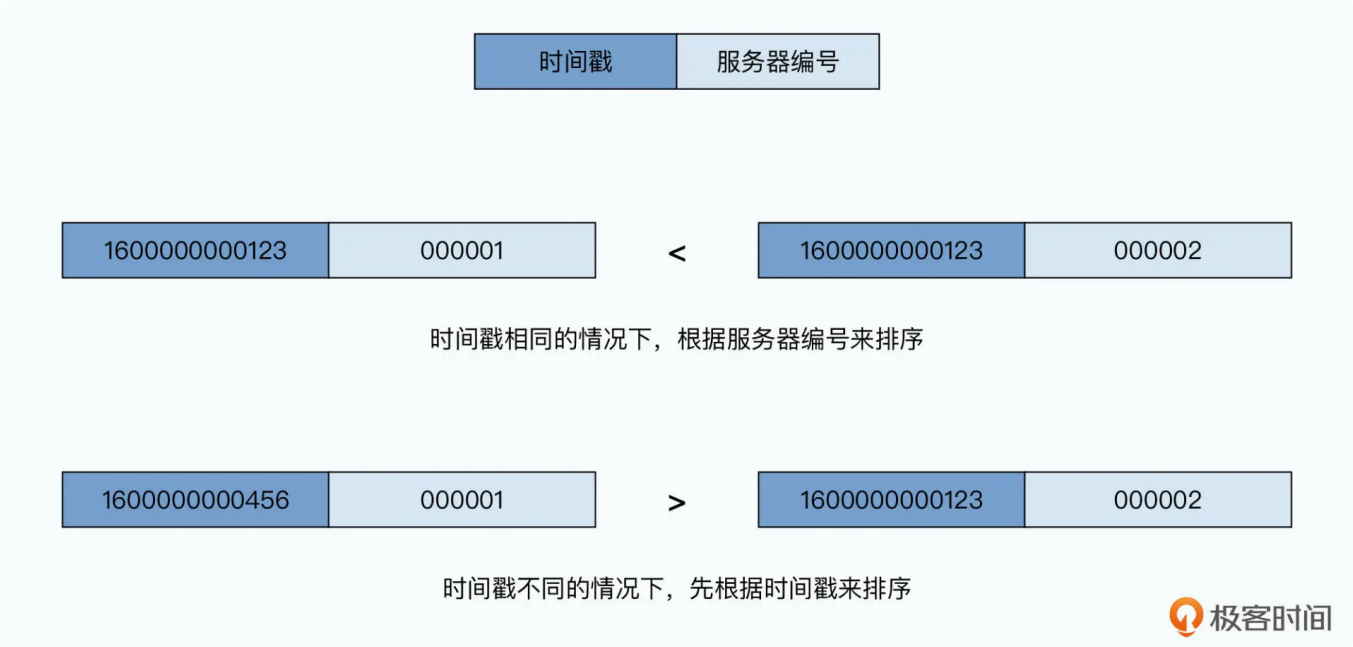
或者，我们可以直接给要更新的记录加锁，在修改完之前，其他人都不能修改对应的数据记录。但是单单读取数据是可以的，不过同样的，读取数据只能读取上一个已经提交了的事务，正在进行的事务是不能读到的。所以，这个方式我们同样也需要对数据加上“版本”信息。

如果是一个单机的事务环境，这个问题很好解决，我们在数据库上通过一个自增的 ID，就可以维护版本了。每当一个事务请求过来，我们就对事务 ID 自增 1，作为新的事务 ID 分配给这个事务。但是**在分布式的环境下，我们同时要读写两台服务器，我们怎么让这两台服务器的共用一个自增的版本呢？**

一种解决方案，是再去搭建一个中心化的事务 ID 生成器，不过这样的话，我们的问题又绕回来了。我们原本为了解决性能问题，让系统变成分布式的了，但是如果有一个中心化的事务 ID 生成器，我们又要面临单点的性能瓶颈、故障风险，以及跨数据中心的网络传输延时问题了。

所以，一般来说，在分布式系统里我们会使用一个天然的信息作为版本，那就是**时间戳**。我们直接用事务提交的时间戳，来作为事务的版本信息，提交得早的，事务 ID 就小。这个也符合我们认知里的自然规律，因为很多所谓的并发，我们只要把时间粒度切到足够细，总也还能分出一个先后来。

不过，计算机的时间精度是有限的，**如果两个事务提交的时间戳一样怎么办呢？**其实这个问题也并不困难，我们只要把所有的服务器编个号，然后把时间戳 + 服务器编号组合在一起，作为事务 ID 就好了。在时间戳相同的情况下，编号小的服务器发起的事务 ID 就更小，我们认为它更早提交就行了。



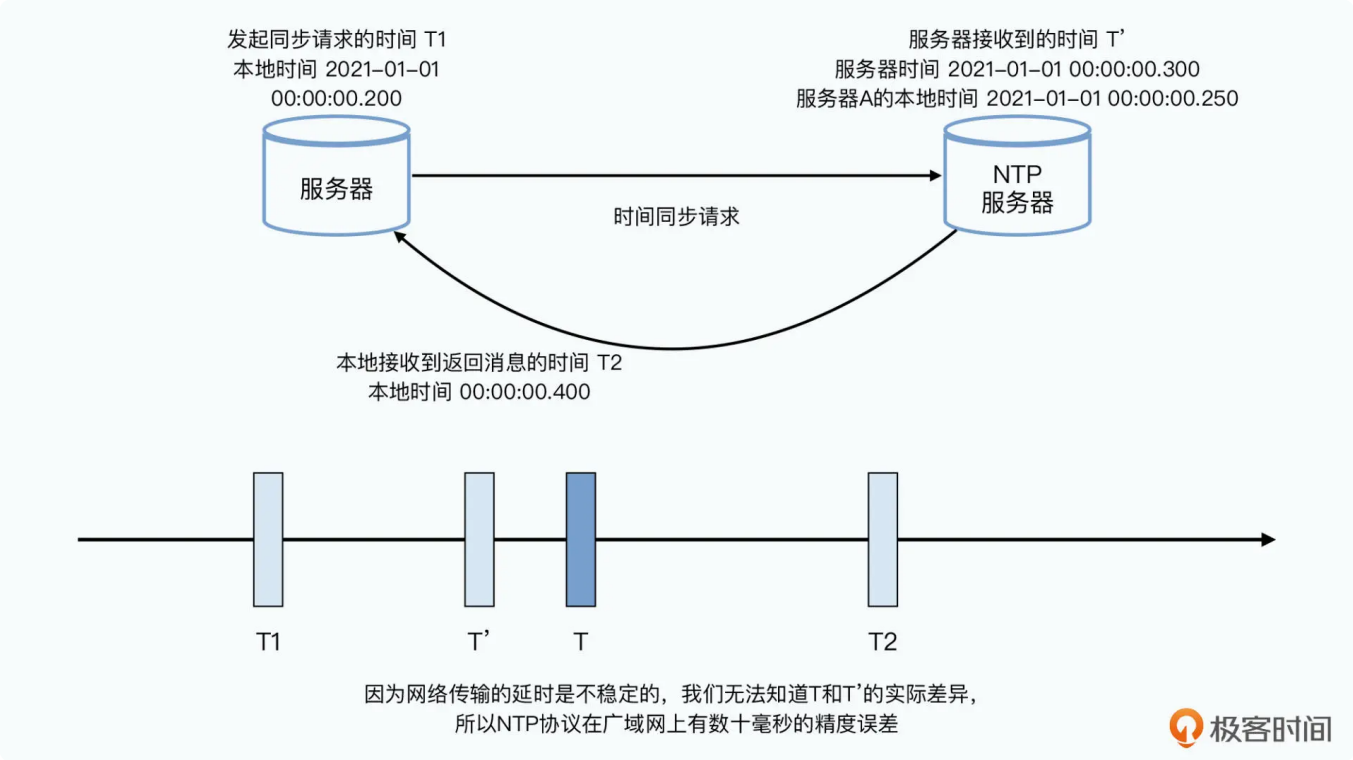
这么一看，Spanner 的分布式事务也并不难处理嘛。不过，当你进入一个真实的世界，而不是只考虑抽象的模型的时候，问题就来了。

如果我们要以时间戳作为事务版本的话，我们能确保所有服务器的时钟是同步的吗？如果我们保障不了这一点，那我们应该用提交事务的服务器的时间戳？还是用实际存储了数据副本的服务器的时间戳？如果它们的时间戳不一样，我们的数据库事务会出现问题吗？

石英表的精度问题

首先，**服务器之间的时钟，是做不到 100% 的完全同步的。**如果你做过 Linux 服务器的安装或者运维工作，你就会知道我们通常会配置一个 NTP 服务器，来同步服务器的时间。服务器里所使用的时钟，是通过石英振荡器来计时的，也就是和我们日常使用的电子石英表是一样。但是它是有误差的，一天的误差会在 1 秒上下。

虽然我们可以频繁地通过 NTP 协议，去进行时钟同步，但是**由于网络延时**的存在，我们也只能把这个误差，做到几十毫秒。你想一下，我们从本地服务器，向一个中心化的远程 NTP 服务器发起请求，我们只能知道，远程的服务器的时间戳 T，我们发起请求的时间戳 T1 和接收到返回的时间戳 T2。



其中，T 是按照 NTP 服务器的时钟计的时，而 T1 和 T2 是按照我们本地服务器的时钟计的时。我们预计远程接收到我们的请求的时间在 T1 和 T2 之间，是一个 T’ ，然后我们需要纠正 T 和 T’ 的时间差，但是我们没有办法精确知道 T’ 到底是多少。考虑到网络延时受到光速的限制，以及公网上传输的延时，比如上海到旧金山可能往返一次花了 150 毫秒。我们只能估计一个 T’ ，最终实际同步完的实际误差，还有可能在几十毫秒。

分布式事务的时间悖论

几十毫秒的时间，对于个人使用 PC 问题不大，但是在 Spanner 这样的上百个数据中心，万亿条数据的分布式数据库，就是一个大麻烦。我们一起来看看，这个时钟的误差，会对前面转账的例子有什么影响。

其实仔细想一下，对于分布式事务选用的时间戳，我们其实没有很好的选择。在单机的数据库事务里，我们可以用实际事务提交的时间戳，比如事务日志写入的时间。但是，**在分布式环境下，不同节点的时间误差，会给我们造成很多麻烦。如果我们选用各个参与者服务器本地的时间戳，就会遇到事务版本不一致的问题。**

我们来看看前面转账的例子：

事务一，张三账户里有 1000 元，要转账 500 给李四。

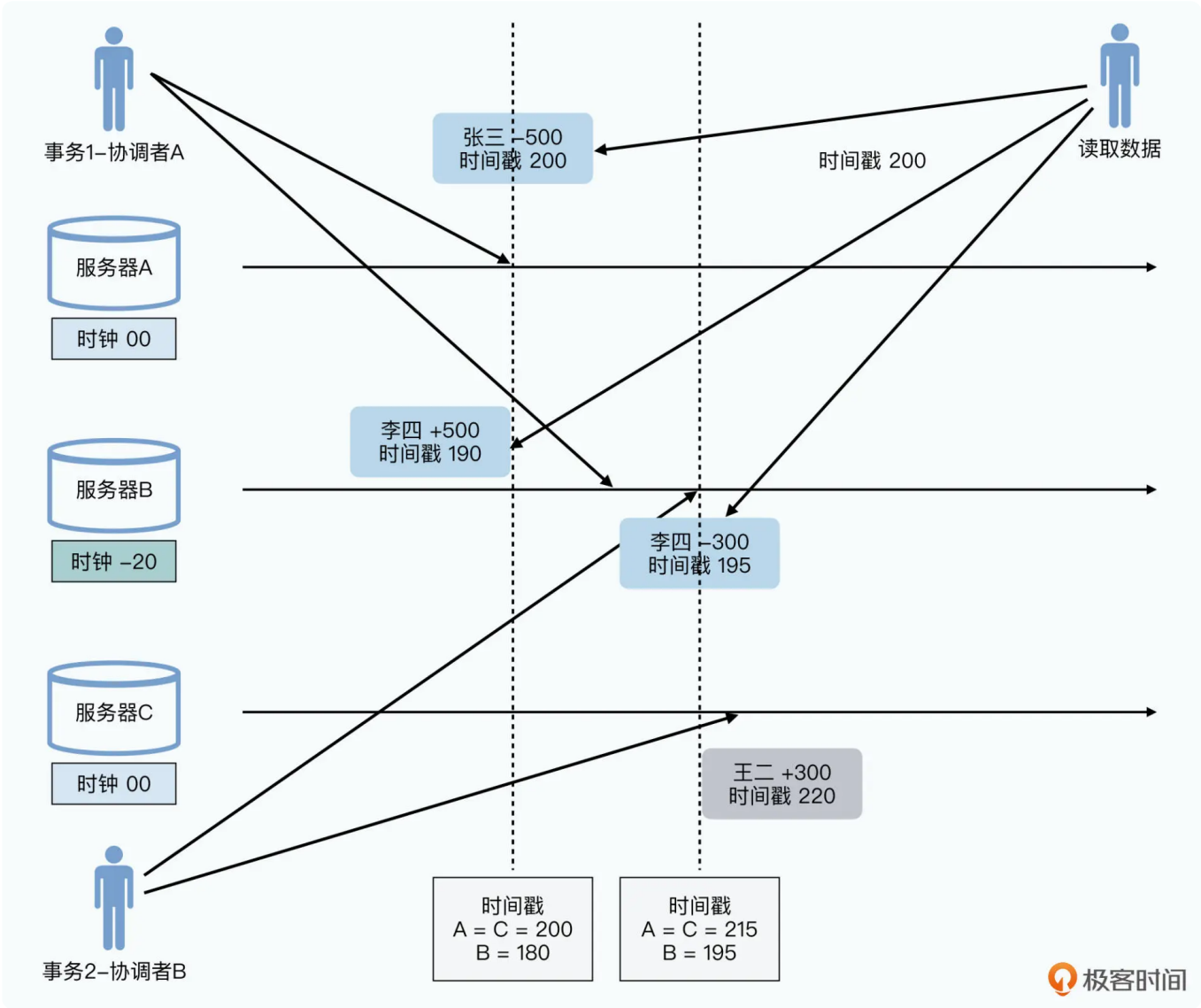
事务二，李四账户里有 500，要转账 300 给王二。

在事务一里，我们要更新 A 里面张三的数据，以及 B 里面李四的数据。事务一，和事务二，实际上在同一秒发起。所以具体时间我们就不列了，张三的数据，在这一秒的第 200 毫秒更新完成了，但是李四的数据，需要通过网络发送到 B，那么就会在这一秒的第 210 毫秒完成。我们把这两个信息都计入到了数据库。

但是 B 的服务器时钟和 A 有误差，差上了 20 毫秒。所以，实际上 B 记录的是事务一，在 190 毫秒就完成了。

然后事务二是从 B 发起的，B 在本地的第 195 毫秒就写完了数据，对应的是 A 的第 215 毫秒。然后向 C 发起了请求，在 C 这边写入了数据，C 和 A 没有时钟差异，所以记录的是 C 的第 220 毫秒。

这个时候问题来了，当我们要去读取数据的时候，我们会发现，那个时间戳是 200 毫秒去读取数据的快照。事务一是完整的，但是事务二里，李四账户里的钱已经没有了，但是还没有出现在王二的账户里。我们的数据是不一致的。



不同服务器之间的时钟差异，使得我们使用服务器的本地时间戳，会带来相同事务的数据版本不一致的问题

那么换一种思路，我们在事务请求里，直接带上发起事务的服务器的时间戳，用这个时间戳行不行呢？我们至少可以保障更新多个不同服务器里的数据的时候，它们的时间戳是一致的。

我们还是回头来看看前面举的转账的例子：

事务一，张三账户里有 1000 元，要转账 500 给李四。

事务二，李四账户里有 500，要转账 300 给王二。

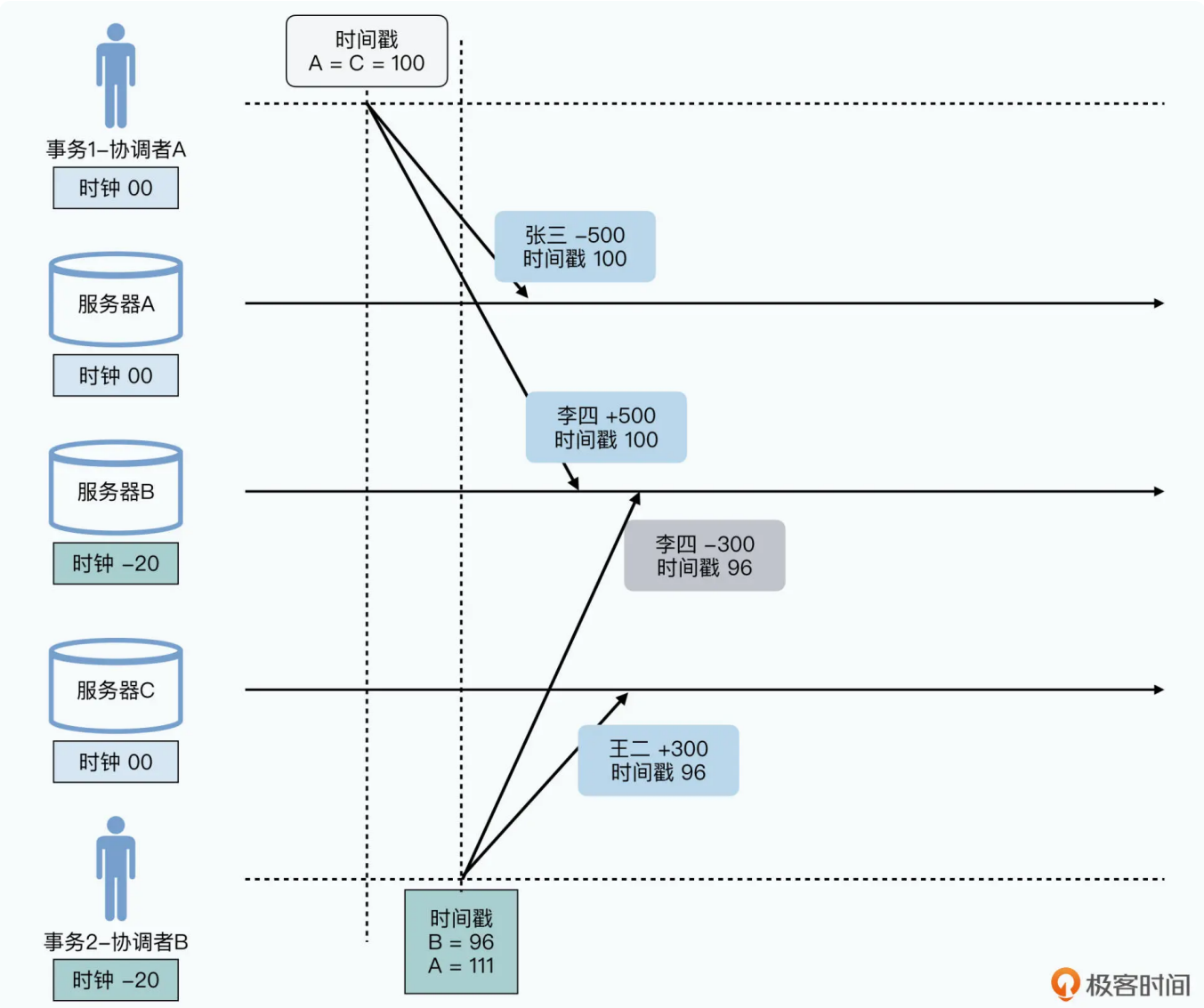
事务一呢，是在这一秒的第 100 毫秒发起的，需要花费 10 毫秒执行完成。事务二呢，是在这一秒的第 111 毫秒发起的，需要花费 10 毫秒执行完成。如果我们没有时间不同步的问题，那么，我们会先执行完事务一，再执行事务二，一切都很完美。

不巧的是，事务二所在的服务器 B，时钟比事务一的服务器，要差上 15 毫秒。这也就意味着，当事务二的请求发出的时候，它带上的时间戳，不是 111 毫秒，而是 96 毫秒。当这个请求到达服务器 B 的时候，问题就出现了。

事务一已经写入成功了，对应写入数据的时间戳，是 100 毫秒。

事务二在这个时候，要读取最新的数据快照，那么它应该使用哪一个时间戳作为版本来读取这个快照呢？

- 如果采用事务请求里的 96 毫秒，那么这个数据库事务就读不到最新的数据。如果我们允许事务提交，就会错误覆盖数据库里的数据。
- 如果我们读取最新的 100 毫秒的数据，那么一旦要提交的时候，就会发现数据里已经有一个 100 毫秒的更新的版本了，也一定提交不成功。

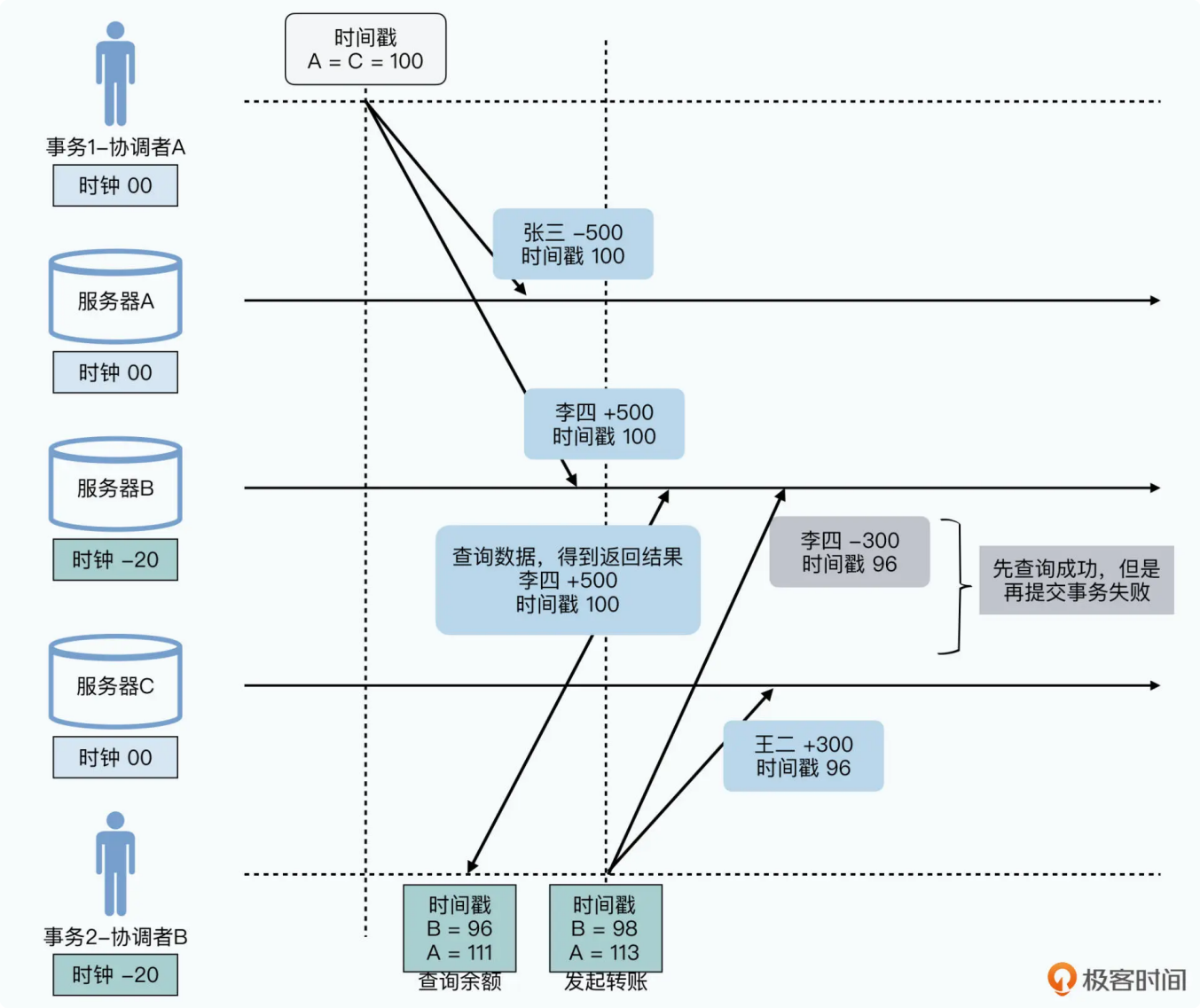


时钟差异，会导致后提交的事务，无法正常提交完成

那么，我们能不能让协调者先获取参与者的时间，用参与者里面时钟比较晚，也就是比较快的那一个呢？如果上面的事务 2，如果我们使用比较快的参与者 C 的时钟，是不是就没有这个问题了？

其实问题还是一样的，如果 C 的时钟和 B 一样，慢上 20 毫秒，我们说的问题仍然存在。并且，这个还是两个事务之间读写的数据有交集的情况下。如果我们我们把事务二，拆分成两个独立的操作，问题就会显得更加奇怪了。

1. 首先，事务一已经写入成功了，对应写入数据的时间戳，是 100 毫秒。
2. 这个时候，张三通过其他渠道，告诉李四已经转账成功了，比如，李四收到了一条短信。
3. 然后是事务二，我们拆分成两个事务，从 B 发起分别执行：首先是，查询数据库里张三的余额，时间戳是 96 毫秒；查询到余额之后，再发起一次转账，时间戳是 98 毫秒。



其实不只是事务层面，我们考虑整个事务发生的“因果顺序”，就会发现时钟的不一致使得我们在开发过程中，会遇到时间错乱的情况

这个时候，我们就面临一个问题了，事务二拆分后查询余额的操作，应该查询到什么数据呢？如果查询失败，那么对于用户来说就会很奇怪，明明已经转账现实成功了，却查询不到。

而如果我们直接无视查询请求的时间戳，直接找到最新的版本，那么后面发起转账的事务操作就会变得奇怪，因为发起转账操作的时间戳比事务一的小，所以事务会失败。但是明明我们刚刚查询过银行卡里有余额，但是实际事务提交的时候却失败了，也很奇怪。

这些失败，带来的结果就是在分布式系统下，我们无法做到系统的“可线性化”。这个“可线性化”，其实就是 Spanner 论文里说的“**外部一致性**”（external consistency）。

尽管在事务层面，我们可以简单地让事务失败、重试来保障数据库本身的“可串行化”。但是因为真实世界的“时间”是客观存在的。我们会遇到，在 A 服务器已经提交成功了，但是因为 B 服务器的时钟和 A 的不一致，再从 B 服务器去查询、或者提交事务的时候，事务仍然会失败的情况。

这个会让用户和开发人员非常困惑，而且当这些问题出现的时候，因为问题和时钟相关，我们也很难 debug，分不清楚到底这个是因为时钟差异导致的正常情况，还是我们系统中的确有明显的 Bug。

原子钟和置信区间

那么，Spanner 是怎么来解决这个问题的呢？其实，本质上也只有两个核心点。**第一个，是尽可能缩短服务器之间的时钟差异；第二个，则是在缩短了时钟差异之后，让所有的数据写入，不再是有一个时间戳，而是给出一个时间戳的范围。**这个时间戳范围，我们可以认为是准确的，因为我们已经通过前面缩短时钟差异的方法，确保服务器之间的时钟差异，不会超过这个范围。而通过这个范围，我们进行一个“有条件的等待”，确保事务提交的先后顺序关系。

原子钟和 GPS

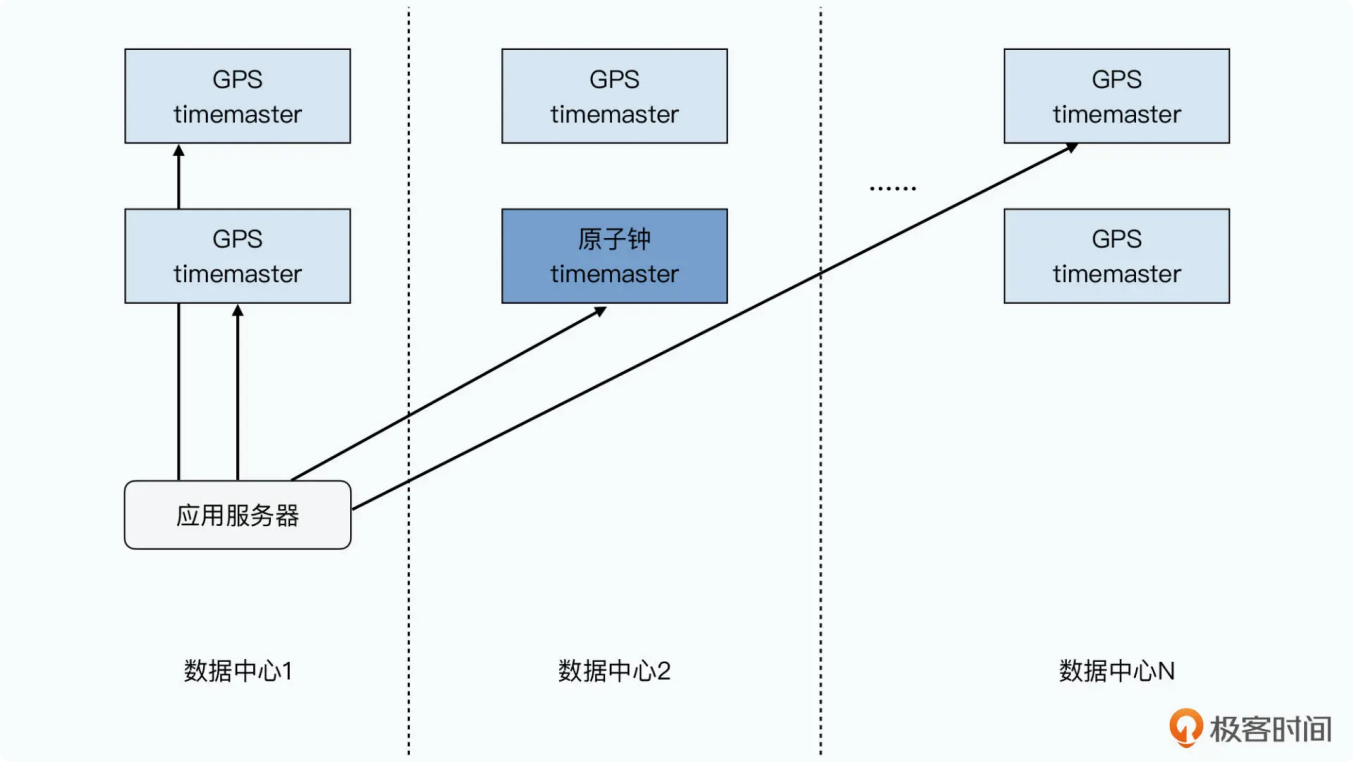
缩短时钟差异的办法，本质上就是**更换时钟同步的方式**。Google 在 Spanner 上，是选择使用原子钟和 GPS 的组合。

原子钟，是根据本地的同位素放射的固定频率来计时，所以可以做到在世界各地的时间是同步的。而 GPS 时钟，则是通过接收卫星信号，计算出本地的时间。它们都利用了物理学规律来计算时间，而不需要担心广域网上不稳定的网络传输带来的延时问题。

选用两种硬件组合的原因，也是**为了“容错”**。原子钟和 GPS 都有可能出现故障，但是出现故障的原因不一样，GPS 可能会因为天线和接收器失效之类出现故障。而原子钟也会出错，但是和外部的天线接收器之类无关。通过两个独立没有关联的系统互为备份，使得整个系统失效的概率就降低了。

Google 在每个数据中心里，都会部署一些 timemaster 机器，大部分的 timemaster 上都会安装 GPS 天线和接收器，还有一些 timemaster 则安装了原子钟。timemaster 之间会互相校验时间，如果某个 timemaster 发现自己的本地时间，和其他 timemaster 相比差异很大，它就会主动下线，把自己设置成离线状态。

原子钟会定期广播一个逐步增长的时间漂移，确保其他服务器能够知道随着时间不断过去，原子钟也慢慢会有时间误差。当然，定期原子钟会同步一次时间，把这个漂移重置为 0。而 GPS 时钟，则会广播一个时间上的不确定性（Uncertainty），一般来说这个数据基本接近于 0。



应用服务器不会只依赖本地数据中心的GPS时钟和原子钟，还会查询其他数据中心的timemaster，来确保“容错”能力

数据中心里的其他服务器呢，则会查询多个 timemaster 进行时钟同步，有些是本数据中心的，有些是其他数据中心的；有些是 GPS 时钟，有些则是原子钟。这个也是为了容错，避免某个 timemaster 或者某个数据中心的数据不准确而影响结果。

条件等待策略

通过 GPS 时钟和原子钟，Spanner 可以把时钟误差范围缩小到 1 毫秒到 7 毫秒之间，平均是 4 毫秒，这个已经比之前的 NTP 时间服务器同步的方式下降了一个数量级。但是，时钟误差始终是存在的，那么我们就要在事务机制上解决这一点。

Spanner 的核心想法很简单，既然我们可以几乎保证，所有的时钟误差都在 7 毫秒以内，那么最坏情况，无非是我们提交的事务，去等 7 毫秒就好了。Spanner 的基本策略是这样的：

> 首先，我们不再只依赖“协调者”的时间了，而是同时考虑所有事务参与者的服务器的时钟。
>
> 所以，Spanner 在提交事务的时候，不仅要看协调者的本地时间，还要知道所有事务参与者中最大的本地时钟。
>
> 其次，Spanner 并不是看此时此刻某一台的时间戳是多少，而是**给每一个事务请求分配一个时间戳**。这个分配的时间戳，需要确保晚提交的事务的时间戳，一定比早提交事务的时间戳晚。
>
> 当事务实际提交的时候，Spanner 会去对比这个分配的时间戳，和本地的时间。如果本地的时间，算上误差，也还没有到分配的时间，事务会一直等待在那里。等知道这个时间过去之后，事务才会实际提交。

所以，Spanner 其实是**在所有的服务器上增加了一个中间层**，在两阶段事务的时候，通过协调者，收集所有事务参与者本地的时间。然后推断在“最坏”情况下，这个事务在什么时候完成，一定不会早于实际可能的时间，并且把这个作为事务的实际的时间戳。

小结

好了，到这里，你就对 Spanner 的分布式事务面临的挑战和 Spanner 解决问题的思路有所了解了。

在 Spanner 这样的全球数据库里，我们想要实现事务就要面临很多过去意想不到的麻烦。这个其中的关键点，就是**跨数据中心的网络延时**和**服务器时钟不同步**的问题。我们看到，传统的 NTP 同步时钟的方式，会有数十毫秒的时钟差异。这个给我们的分布式事务带来了很大的挑战。我们没有中心化的协调者，也无法简单使用事务参与者的时间戳。而不同的协调者的时钟差异，又使得我们的事务操作，无法做到“可线性化”。

所以，Google 采用了**原子钟 +GPS 时钟**，来缩短各个服务器的时钟差异问题。不过，时钟差异只能缩短，不会消失。但是时钟差异大大缩短之后，这个分布式事务的“可线性化”问题已经有工程上的解决办法了。

Google 的策略也很简单，就是让协调者，收集所有事务参与者的本地时钟，以及它们可能的误差。然后综合所有的时间信息，为事务分配一个时间戳。而实际事务提交的时间，则是在各个机器上，都等待到这个时间戳过去之后。因为时间差异是有限的 7ms，那么最坏情况下，无非也就是我们的事务需要多等待 7ms。

那么，下节课里，我们就深入来看看，这个综合所有参与者的时间信息，以及分配时间戳的解决方案具体是怎么样的，它的 API 是什么样的，在事务并发的情况下，会对 Spanner 的性能又有什么样的影响。

推荐阅读

在分布式系统里，最大的挑战就是在于部分系统故障和失效的情况。其中，无论是网络还是时钟都是不可靠的。我们无法保障跨地域的数据中心里不同的服务器的时钟是一致的，我们也无法确定，跨数据中心的网络请求能够在一个确定时间内到达。那么对于这个问题，如果你想深入了解，可以去读一读《数据密集型应用系统设计》的第 8 章。

此外，如果你对我们这节课所说的时钟不同步的问题，想要自己研究一下的话。这个链接里，有一系列关于计算机时钟的文章，非常值得一读。

思考题

在这节课里，我们看到 Spanner 想要的“外部一致性”，其实就是我们之前聊过的“可线性化”。除了我这里举的例子之外，你能想想还有什么其他例子，也会遇到这种时钟差异问题带来的奇怪体验吗？你自己在日常使用各种互联网产品的时候，遇到过这样的情况吗？

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | Spanner（一）：“重写”Bigtable和Megastore

下一篇 24 | Spanner（三）：严格串行化的分布式系统

精选留言 (2)

写留言

在路上

2021-11-22

徐老师好，看直播的时候有时候会卡顿，当我重新连接服务器之后，看的内容不是最新的，而是掉线前看过的内容。如果两个人同时用手机看直播，很可能出现一个人看到的内容落后另一个人看到的内容，那个看到落后内容的人要么会带上耳机，假装别人正在看的直播不存在，要么关掉自己的手机，凑过去和别人一起看。

展开

2

那时刻

2021-11-22

处理时间的问题一致是痛点，其一是时间同步；其二是处理时区的时间。